QEMU-Based Emulation-in-the-Loop for the Simulation of Small Satellite Flight Software

Rachel Misbin
NSF SHREC Center
University of Pittsburgh
Pittsburgh, PA 15260
rachel.misbin@nsf-shrec.org

Alan George
NSF SHREC Center
University of Pittsburgh
Pittsburgh, PA 15260
alan.george@nsf-shrec.org

Abstract—**Hardware-in-the-loop testing is a popular, economical way to test and validate flight software systems for small satellites compared to testing on flight hardware. Further cost savings can be achieved by replacing the hardware in these systems with functionally equivalent emulations (sometimes termed "emulation-in-the-loop"). This work aims to better inform system-level small satellite flight software test design through a study of the tradeoffs between hardware-in-the-loop and emulation-in-the-loop systems. In this work, two systems are presented—one featuring hardware, the other featuring emulation—which are used to evaluate the advantages and disadvantages of a traditional hardware-in-the-loop testbed compared to an emulation-in-the-loop testbed. Two demonstrations are performed on these two testbeds: (1) a large-scale spacecraft cluster demonstration, which seeks to show how hardware and emulation perform at scale; and (2) an image processing demonstration, which seeks to show how hardware and emulation differ for more compute-intensive applications. The hardware-in-the-loop testbed is composed of an instance of NASA's core Flight System (cFS) running on the ARM Cortex-A9 processor of a Zynq-7020 System-on-Chip (SoC) interfaced with NASA Goddard's open-source 42 spacecraft simulator to create a closed-loop system. The emulation-in-the-loop testbed is identical but with the ARM processor emulated instead. Emulation-in-the-loop systems offer portability, cost savings, and improved scalability over hardware-in-the-loop systems but at the cost of system accuracy and increased complexity. In the case of flight software for small satellites, the inherent reduced accuracy of emulation-in-the-loop can prove acceptable in light of the significant cost savings, the reduction in dependence on hardware early in development, and the reduction of wear on hardware. In addition to the two emulation-based and hardware-based demonstrations, a comprehensive view of the benefits and drawbacks of QEMU-based emulation as a replacement for traditional hardware is presented for the case of in-the-loop simulation of flight software.**

TABLE OF CONTENTS

978-1-6654-9032-0/23/$31.00 ©2023 IEEE

1. INTRODUCTION

Despite the move to incorporate more commercial-off-the-shelf hardware, space test hardware remains expensive and difficult to source. Often, as software for a spacecraft is being developed, the processors that will run this software remain inaccessible for use across a team due to the high cost of hardware, the fragility of such hardware, and the difficulty of replicating hardware environments at scale. This problem is exacerbated for applications involving multiple spacecraft, where the desire to perform in-the-loop software development and testing is further hindered by the cost and inaccessibility of avionics hardware.

Recent advances in open-source emulation technologies hint at the possibility that emulation could provide earlier, less expensive in-the-loop testing for small spacecraft than can traditionally be done with hardware. This work aims to explore the advantages and disadvantages of a QEMU-based emulation-in-the-loop setup compared to a more traditional hardware-in-the-loop setup for the simulation of small satellite flight software. Two experiments are performed to evaluate these two testbeds: the first involves scaling each testbed to approximately a dozen spacecraft and the second involves performing compute-intensive image processing on each testbed.

First, related work on emulation and in-the-loop testing will be presented in Section 2. Then, some background on the various tools and technologies, including hardware-in-the-loop testing, NASA's cFS framework, used in this work will be presented in Section 3. In Section 4, the approach to building and conducting the two experiments on the emulation-in-the-loop and hardware-in-the-loop testbeds will be discussed. Next, in Section 5, the results of the cluster and compute-intensive demonstrations will be presented. In Section 6, these results will be discussed and conclusions will be drawn. Finally, in Section 8, there will be a discussion of possible extensions of this work.

2. BACKGROUND

This research, funded by the NSF SHREC Center, features a number of concepts and technologies including hardware-in-the-loop testing, NASA's cFS framework, the 42 dynamics simulator, and hardware emulation using QEMU. Now, we will discuss these concepts in more detail.

NSF Center for Space, High-performance, and Reconfigurable Computing (SHREC)

This research was conducted at the NSF SHREC center at the University of Pittsburgh. The SHREC center is an Industry-University Cooperative Research Center (IUCRC) program at the National Science Foundation (NSF). SHREC is supported

2023 IEEE Aerospace Conference | 978-1-6654-9032-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/AERO55745.2023.10115569

by and collaborates with 34 partners from industry and government, including NASA's Katherine Johnson Independent Verification & Validation Facility (IV&V) which has been a source of collaboration for this work. This partnership enables students at SHREC to perform research at the intersection of academia and industry on topics ranging from computing for space science to high-performance computing to dependable computing for harsh or critical environments.

Hardware-in-the-Loop Testing

Emulation-in-the-loop systems build off the concept of in-the-loop testing, which involves integrating a unit-under-test (be it hardware, software, or something else) into a larger system to exercise its operation in the greater context under which the unit-under-test operates. Hardware-in-the-loop testing is a popular technique for supporting the development and testing of flight software and other embedded software products. The basis of hardware-in-the-loop testing is feeding software controllers signals to simulate conditions that are otherwise difficult to access. Ideally, flight software would be tested during a flight, allowing it to be subjected to real data and conditions. This is clearly impractical and costly, making hardware-in-the-loop testing a valuable alternative. Hardware-in-the-loop testing allows flight software to be exercised in a variety of both nominal and off-nominal cases to improve its robustness and allow developers to engage in the test-and-develop cycle popularized among other software development processes. For instance, hardware-in-the-loop testing allows developers the ability to test software conditions that would otherwise be dangerous or impossible to replicate. This offers developers greater insight on all paths through their software.

Core Flight System (cFS)

Core Flight System (cFS) is a reusable open-source flight software framework developed by NASA to reduce duplication of effort in developing software for small spacecraft [8]. cFS is designed to be lightweight and modular while providing certain reusable core services commonly used by flight software systems.

cFS has been featured on a number of missions conducted by NASA, including the Lunar Atmosphere Dust and Environment Explorer (LADEE), the Lunar Reconnaissance Orbiter (LRO), and the Radiation Belt Storm Probes (RBPS) mission among others. The cFS framework was chosen for this work due to its open-source nature and its flight heritage, representing a realistic tool for the development of flight software systems. Additional benefits of cFS are that it facilitates formalized software reuse across missions, it reduces project schedule, and it simplifies maintenance.

42 Dynamics Simulator

42 is a general-purpose, open-source spacecraft attitude and orbital dynamics simulator developed at the NASA Goddard Space Flight Center to support flight software development [14]. It supports rapid prototyping for flight software and guidance and navigation control design, allowing users to create new sensors, actuators, vehicle geometries, and more. 42 also features a graphical mode using OpenGL for graphics output which can render a view of the spacecraft or a view from the spacecraft. For this work and other demonstrations, we extended 42's render capability to create a camera view, acting as a type of sensor capable of capturing images on command and sending them over an independent socket to external flight software. Figure 1 shows the 42 spacecraft view alongside the spacecraft camera view (boxed in red).

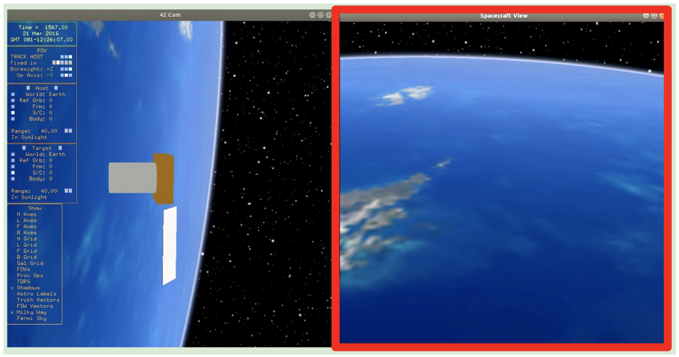
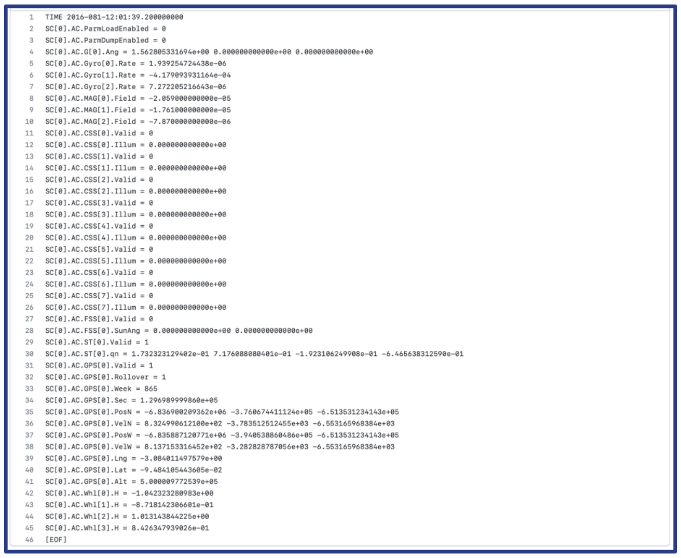


Figure 1. 42 spacecraft view (left) and spacecraft camera view (right)



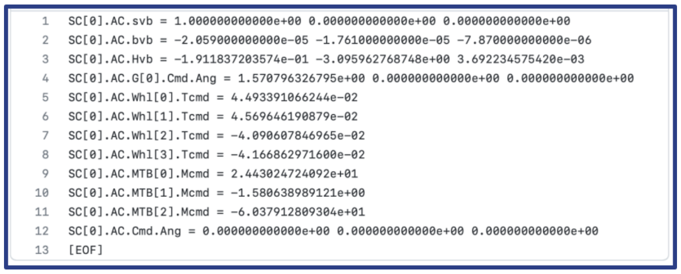
```
TIME 2016-081-12:01:39.200000000
SC[0].AC.ParmLoadEnabled = 0
SC[0].AC.ParmDumpEnabled = 0
SC[0].AC.G[0].Ang = 1.562805331694e+00 0.000000000000e+00 0.000000000000e+00
SC[0].AC.Gyro[0].Rate = 1.939254724438e-06
SC[0].AC.Gyro[1].Rate = -4.179093931164e-04
SC[0].AC.Gyro[2].Rate = 7.272205216643e-06
SC[0].AC.MAG[0].Field = -2.059000000000e-05
SC[0].AC.MAG[1].Field = -1.761000000000e-05
SC[0].AC.MAG[2].Field = -7.870000000000e-06
SC[0].AC.CSS[0].Valid = 0
SC[0].AC.CSS[0].Illum = 0.000000000000e+00
SC[0].AC.CSS[1].Valid = 0
SC[0].AC.CSS[1].Illum = 0.000000000000e+00
SC[0].AC.CSS[2].Valid = 0
SC[0].AC.CSS[2].Illum = 0.000000000000e+00
SC[0].AC.CSS[3].Valid = 0
SC[0].AC.CSS[3].Illum = 0.000000000000e+00
SC[0].AC.CSS[4].Valid = 0
SC[0].AC.CSS[4].Illum = 0.000000000000e+00
SC[0].AC.CSS[5].Valid = 0
SC[0].AC.CSS[5].Illum = 0.000000000000e+00
SC[0].AC.CSS[6].Valid = 0
SC[0].AC.CSS[6].Illum = 0.000000000000e+00
SC[0].AC.CSS[7].Valid = 0
SC[0].AC.CSS[7].Illum = 0.000000000000e+00
SC[0].AC.FSS[0].Valid = 0
SC[0].AC.FSS[0].SunAng = 0.000000000000e+00 0.000000000000e+00
SC[0].AC.ST[0].Valid = 1
SC[0].AC.ST[0].qn = 1.732323129402e-01 7.176088080401e-01 -1.923106249908e-01 -6.465638312590e-01
SC[0].AC.GPS[0].Valid = 1
SC[0].AC.GPS[0].Rollover = 1
SC[0].AC.GPS[0].Week = 865
SC[0].AC.GPS[0].Sec = 1.296989999860e+05
SC[0].AC.GPS[0].PosN = -6.836900209362e+06 -3.760674411124e+05 -6.513531234143e+05
SC[0].AC.GPS[0].VelN = 8.324990612100e+02 -3.783512512455e+03 -6.553165968384e+03
SC[0].AC.GPS[0].PosW = -6.835887120771e+06 -3.940538860486e+05 -6.513531234143e+05
SC[0].AC.GPS[0].VelW = 8.137153316452e+02 -3.282828787056e+03 -6.553165968384e+03
SC[0].AC.GPS[0].Lng = -3.084011497579e+00
SC[0].AC.GPS[0].Lat = -9.484105443605e-02
SC[0].AC.GPS[0].Alt = 5.000009772539e+05
SC[0].AC.Whl[0].H = -1.042323280983e+00
SC[0].AC.Whl[1].H = -8.718142306601e-01
SC[0].AC.Whl[2].H = 1.013143844225e+00
SC[0].AC.Whl[3].H = 8.426347939026e-01
[EOF]
```

Figure 2. Example sensor message sent from 42 to cFS

Most important to this work, 42 is very extensible and can be connected to external, or standalone, flight software in a relatively simple way. For this work, 42 is used with standalone flight software (a set of cFS apps) interfaced over sockets. One socket allows standalone flight software to exchange sensor and actuator commands. Structured commands and data are sent and received at regular intervals. Thus, the program flow after initialization consists of 42 sending a message containing sensor data to the standalone flight software, then the flight software performing any necessary processing or interpretation of that data and responding with a message containing actuator commands. An example of a sensor message (termed a SensorMsg) and an actuator message (term an ActuatorMsg) are shown in Figures 2 and 3, respectively.

QEMU and Hardware Emulation

QEMU (Quick EMUlator) is a free and open-source hardware emulation tool used widely across industry and academia. It is capable of fast machine emulation using a portable dynamic translator. It has full system support for many popular embedded and desktop platforms and is designed to allow developers to create new models to support additional platforms [1]. QEMU consists of a CPU emulator, emulated devices, generic devices, machine descriptions, a debugger, and a user interface. The core of QEMU's functionality



```
SC[0].AC.svb = 1.000000000000e+00 0.000000000000e+00 0.000000000000e+00
SC[0].AC.bvb = -2.059000000000e-05 -1.761000000000e-05 -7.870000000000e-06
SC[0].AC.Hvb = -1.911837203574e-01 -3.095962768748e+00 3.692234575420e-03
SC[0].AC.G[0].Cmd.Ang = 1.570796326795e+00 0.000000000000e+00 0.000000000000e+00
SC[0].AC.Whl[0].Tcmd = 4.493391066244e-02
SC[0].AC.Whl[1].Tcmd = 4.569646190879e-02
SC[0].AC.Whl[2].Tcmd = -4.090607846965e-02
SC[0].AC.Whl[3].Tcmd = -4.166862971600e-02
SC[0].AC.MTB[0].Mcmd = 2.443024724092e+01
SC[0].AC.MTB[1].Mcmd = -1.580638989121e+00
SC[0].AC.MTB[2].Mcmd = -6.037912809304e+01
SC[0].AC.Cmd.Ang = 0.000000000000e+00 0.000000000000e+00 0.000000000000e+00
[EOF]
```

Figure 3. Example actuator message sent from cFS to 42

comes from its dynamic translator, which accepts instructions for the target CPU and transforms them into instructions for the host at runtime.

The idea of hardware emulation is to replicate the behavior of one computer system on another, often for testing and development. For instance, if a software developer is creating a program to execute on an ARM architecture but does not have access to ARM hardware, they could run their program in an emulated ARM environment. Furthermore, hardware device emulation can allow developers to specifically target platforms with completely different instruction set extensions, compatible execution modes across adjacent families, and more. Software development requires many iterations and tests, so it is an advantage for developers to be able to easily test their work on the specific hardware platform it is ultimately designed to work on. Otherwise, porting software from one architecture to another often necessitates modifications to ensure correct and efficient operation (if it's possible in the first place). Thus, hardware emulation is a logical tool to use when a platform is costly or otherwise difficult to access, as is often the case in flight software development.

In fact, a study commissioned by the NASA Office of Chief Engineer specifically highlights the high time and developer cost as well as engineering and project costs that often result due to delays in flight software development [4]. Hardware emulation offers a ready-to-go solution to testing flight software which is far faster and easier than working with hardware development kits.

3. Related Work

This research was inspired by several other efforts related to the use of QEMU to replace hardware for in-the-loop testing and for flight software demonstrations. This includes the emulation of cyber-physical systems using QEMU, NASA's Simulation-to-Flight-1 mission, an effort to emulate an internet of things system using QEMU, some work on software reliability for small satellites, and several flight software demonstrations using 42 and cFS.

Emulation-in-the-Loop Testing with KhronoSim

Emulation-in-the-loop studies are relatively new but there do exist several notable examples. In 2018, a team of researchers from the CISTER Research Center published a study of emulation-in-the-loop for the testing of real-time critical cyber-physical systems [10]. This involved the development of a platform called KhronoSim, designed to perform closed-loop testing of cyber-physical systems through the use of models and emulated hardware. The goal of developing and using KhronoSim was to serve as a platform for testing different design choices before committing to hardware to accelerate the future development of similar projects.

In addition to describing the architecture and features of KhronoSim, this work explores some of the advantages of using hardware emulation over physical hardware as well as an overview of emulator choices the authors investigated. Ultimately, QEMU was selected for its support of many devices as well as its stability and community support relative to other emulators that were evaluated.

To interact with QEMU, they developed a monitor on top of QEMU's Machine Protocol (QMP) which uses sockets to send and receive commands to and from each QEMU instance. Through this interface, they were able to make use of QEMU's throttle control functionality to align the timing of the emulator with other modules in their testbed.

Simulation-to-Flight-1 (STF-1) Mission

The STF-1 mission involved designing, constructing, and testing a small satellite and was built by NASA's Katherine Johnson Independent Verification & Validation Facility in conjunction with West Virginia University [9]. STF-1 made use of many of the same tools that are used in this work, including the 42 simulator and cFS. Additionally, this project presented a study of the NASA Operational Simulator for Small Satellites (NOS^3), a tool designed to enhance software-only simulation capabilities for small satellite missions, by showcasing its use in the STF-1 mission. NOS^3 consists of a simulator architecture interfaced with a number of different software- and hardware-based models based around the NASA Operational Simulator (NOS) engine middleware.

The ultimate goal of the STF-1 mission was to demonstrate the value of software-based simulation and testing for small satellite missions. By integrating the flight software for STF-1 with NOS^3, researchers were able to show the value of software-only simulation for the development and operation of a small satellite. The project involved considerable flight software development, at more than 132,000 source lines of code (SLOC; a measure of total lines of code excluding whitespace and comments), twenty-five percent of which was developed specifically for STF-1. Researchers identified three key benefits of NOS^3 and by extension software-only simulation for small satellite missions.

First, the use of NOS^3 reduced the project's reliance on hardware during development and integration testing allowing for the development of flight software alongside hardware testing. The second benefit cited is the reduced risk during software development by allowing various members of the team to create testing environments and perform testing that might otherwise not be safe or feasible on hardware. The final benefit was the ability to move up the team's software development schedule.

Emulating the Internet of Things (IoT) with QEMU

Another example of a study assessing the performance of hardware emulation comes from a 2019 master's thesis by Gyokan O. Osman of the University of Gothenburg in Sweden on the emulation of the internet of things using QEMU [11]. Osman's goal was to perform a full-system emulation for a distributed group of embedded devices. To accomplish this, Osman created and configured the nRF51 platform, a system-on-chip (SoC) design using the ARM Cortex M0, for QEMU.

This work was assessed by comparing the performance and functionality of the emulation-based system to a hardware-based system. Osman found that QEMU was capable of running programs for the nRF51 platform as stable as hardware aside from a small number of occasions where issues with poor interface implementations and unnecessary debugging statements caused degraded performance. Additionally, the emulated environment was capable of running a real-time operating system. This result is an important indicator that emulation is a viable replacement for hardware for the testing and development of certain systems.

A key takeaway from this work is that the performance of emulation can vary in a way that's different than hardware. Care might need to be taken to ensure that developers aren't introducing artificial performance restrictions on their emulation models. An additional concern is that emulated devices are perfect, whereas real devices might have timing delays or other imperfect or varying behaviors.

Reliability of Small Satellites

Another relevant work comes from Matthew Grubb at West Virginia University on the reliability of small satellites [7]. The focus of this work is on using simulations for the development and testing of flight software for small satellite missions. Grubb identified that flight software is a critical component of any spacecraft but that it tends to be neglected during spacecraft development, leading to it being undertested. To explore possible improvements in the software development lifecycle for small satellites, Grubb explores the 42 simulator as well as the NOS^3. Grubb used NOS^3 interfaced with 42, cFS, and a ground station tool known as COSMOS to demonstrate of a method of easily deploying, accessing, and configuring the flight software development environment. Ultimately, this work demonstrated the use of these tools on the STF-1 mission; these tools' use for this mission contributed considerably to the mission's success because it identified the need for more software testinbg for small satellites.

4. Approach

This section will discuss the techniques used to develop the emulation-in-the-loop and hardware-in-the-loop testbeds as well as details on how their performance is compared with a cluster scaling test and an embedded image processing test. More specifically, we will discuss the cFS-42 interface which underpins both the hardware and emulation testbeds, the details of each testbed, how timing data was collected for each platform, and the details of each test.

cFS-42 Interface

To enable in-the-loop testing, the 42 simulator had to be interfaced with the cFS flight software such that 42 could send spacecraft sensor data to cFS and cFS could reply with commands. This was done through the creation of a custom cFS app. Conveniently, 42 provides an interface to connect to a standalone application through the use of sockets. Thus, a corresponding interface was developed in the custom cFS app which can send and receive data to and from the 42 simulator for any number of spacecraft. Communication occurs through the use of structured messages, termed sensor messages and actuator messages. Examples of these files are shown below.

The architecture for a single spacecraft interfaced to the 42 simulator is shown in Figure 4 and the architecture for

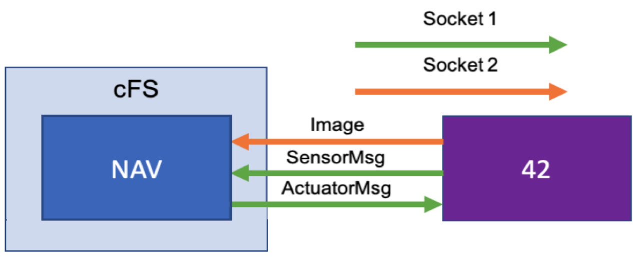


Figure 4. Software architecture for a single spacecraft interfaced to the 42 simulator

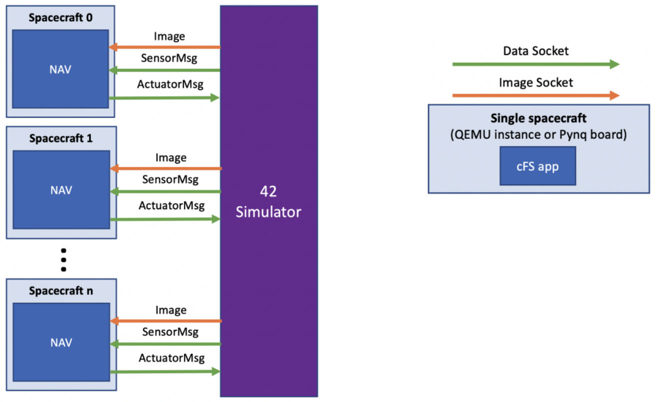


Figure 5. Software architecture for multiple spacecraft interfaced to the 42 simulator

multiple spacecraft interfaced to the 42 simulator is shown in Figure 5.

Hardware-in-the-Loop Testbed

The hardware-in-the-loop testbed is used as a basis for comparison to the emulation-in-the-loop testbed. It features 9 Pynq-Z2 boards, each equipped with a 650MHz ARM Cortex-A9 processor. Pynq-Z2 boards were chosen for this work because they feature the Zynq-7020 System-on-a-Chip (SoC) which have extensive flight heritage, including use in SpaceMicro's CubeSat Space Processor (CSP) [3], SHREC's CHREC Space Processor (CSP) [13], and Xiphos Q7 [12]. This history, in addition to ease-of-access to the Zynq-7020 through SHREC, makes it an obvious choice for this work.

Each Pynq-Z2 board runs a cFS instance interfaced to an instance of 42 running on a host machine via a socket connection over ethernet. Thus, each board represents a "spacecraft" in the in-the-loop system and they each interact with the dynamics simulator independently to represent a cluster of 9 independent spacecraft. The general architecture for the hardware-in-the-loop testbed is shown in Figure 6.

Emulation-in-the-Loop Testbed

The emulation-in-the-loop testbed has a similar architecture to the hardware-in-the-loop testbed but with each Pynq-Z2 board replaced with a QEMU instance. QEMU was chosen to perform hardware emulation for this work due to its flexible, open-source nature. The authors of [10] state that QEMU was selected for their work due to its support for many devices, its stability, its community support, and its throttle control functionality. Similarly, [11] chose to use QEMU due to

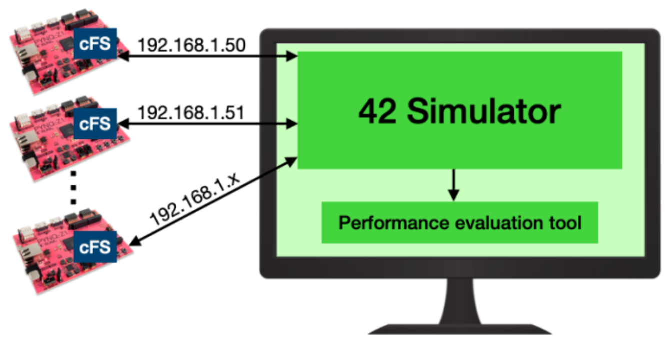


Figure 6. Architecture for hardware-in-the-loop setup

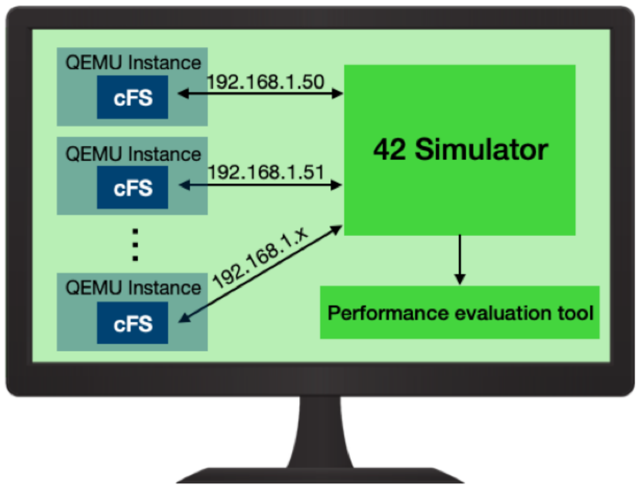


Figure 7. Architecture for emulation-in-the-loop setup

its wide-spread adoption in security research and testing and its history of use for creating software-based prototypes of embedded systems. In addition to its support from related work, QEMU is used at NASA IV&V for their software-based simulation and emulation efforts, making QEMU a natural choice of emulator for this work.

In this research, each QEMU instance is its own process on the host machine, and like in the hardware-in-the-loop testbed, each QEMU instance runs its own instance of cFS. These QEMU instances are interfaced to 42 over their own sockets, much like the hardware-in-the-loop testbed.

The emulation-in-the-loop testbed is run on a desktop workstation similar to what might be used by flight software developers, specifically a 64-bit x86 workstation with an Intel Core i7-6700 CPU operating at 3.4 GHz.

Data Collection

To measure the performance of each test, the time between the reception of Sensor Messages and the sending of Actuator messages is measured over approximately 500 exchanges. This gives an estimate of the impact of emulation rather than hardware on the timing performance of each test. Ideally, the difference in execution time between hardware and emulation should remain zero for each test. Given the relative speed of hardware over software, however, it's expected that hardware will generally be faster than software-based hardware emulation and that the difference in execution time between hardware and emulation will be nonzero and positive. Therefore, this timing data will yield insight on timing performance degradation for the use of emulation over hardware for the two scenarios under test.

Through a comparison of the timing of the emulation-in-the-loop and hardware-in-the-loop testbeds, it is possible to determine how timing performance degrades in emulation-in-the-loop as the cluster size is increased or as the intensity of computation increases.

Cluster Scaling Test

The first test for the hardware-in-the-loop and emulation-in-the-loop testbeds explores the scalability of each platform. The scalability of the hardware-in-the-loop platform is limited to the physical number of boards available (9) so this serves as a basis for comparison. The emulation-in-the-loop testbed, on the other hand, is not restricted by physical hardware but rather by the capabilities of the host computer.

Testbed scalability is an important factor for missions involving spacecraft clusters. To be useful for such missions, an emulation-in-the-loop testbed must be capable of performing on a sufficiently similar timescale to hardware-in-the-loop testbeds even at scale. The goal of the cluster scaling test is therefore to determine at which point the overhead of emulation becomes too large for simulating cluster missions.

Embedded Image Processing Test

The second test for the hardware-in-the-loop and emulation-in-the-loop testbeds explores the capability of independent spacecraft in each testbed when confronted with more compute-intensive tasks such as image processing. For this test, each testbed is configured with two spacecraft; one spacecraft can see, meaning it has a camera integrated and is configured to run an image processing algorithm on every frame it captures, and the other spacecraft does not. The goal of the simulated mission is for the spacecraft with image processing capabilities to detect the other spacecraft.

Contrary to the cluster scaling test, this test seeks to evaluate the impact of more intensive computation on an emulation-in-the-loop testbed. Whereas the cluster scaling test deals with each testbed's ability to handle additional communication between the host machine and individual hardware/emulation instances, this test explores the impact of additional computational overhead on each testbed.

The algorithm used to process images is adapted from a lightweight image processing library developed for image processing on space platforms [5]. The primary compute function is shown in Figure 8.

5. Results and Evaluation

As a general rule, hardware tends to be faster than software, so it was expected that hardware would offer better timing performance than emulated hardware. This is because while the hardware setup only needs to run flight software, the computer running the emulated setup needs to run each QEMU guest instance on top of the existing operating system installed on that computer, complete with context switches and other host-specific operations. In other words, we are looking for the breaking point as a function of scale where emulation can no longer (at all or consistently) keep up with hardware. The results of the cluster scaling test and the

```c
int NAV_LocateRGB(FILE * data, int r, int g, int b, int
↪ threshold)
{
    int i, j;
    int count = 0;
    double x_temp = 0, y_temp = 0;
    struct rgbpixel rowbuffer[10000];

    for (i = 0; i < 800; i++)
    {
        if (fread(rowbuffer, sizeof(struct rgbpixel), 800, data)
        ↪ != (unsigned) 800)
        {
            fprintf(stderr, "Warning: End of input image reached
            ↪ early in colorclassifier\n");
            return (-3);
        }
        for (j = 0; j < 800; j++)
        {
            if ((abs(rowbuffer[j].red - r) <= threshold) &&
            ↪ (abs(rowbuffer[j].gre - g) <= threshold)
                && (abs(rowbuffer[j].blu - b) <= threshold))
            {
                rowbuffer[j].red = 255;
                rowbuffer[j].gre = 0;
                rowbuffer[j].blu = 0;
                x_temp += (double) j / 800;
                y_temp += (double) i / 800;
                count++;
            }
        }
    }

    if (count != 0)
    {
        x_temp = x_temp / count;
        y_temp = y_temp / count;
    }

    return ((x_temp > 0) && (y_temp > 0)) ? 1 : 0;
}
```

Figure 8. The NAV_LocateRGB function serves as the compute-intensive component of this test. It performs computations over the entire image using nested for-loops to locate the RGB value passed to the function

embedded image processing test are discussed below.

Cluster Scaling Test

As shown in Figure 9, the time between receiving sensor data and sending actuator commands increases approximately linearly with the number of spacecraft in the emulation. As a result, it might not be ideal to emulate large spacecraft clusters requiring strong timing precision without additional optimization to the emulation environment or its interfaces. Much of this overhead likely results from the increased socket communication and computation required for each additional instance, similar to the limitations found in [11].

This indicates that unless modifications are made, for large cluster sizes, emulation-in-the-loop introduces approximately twice the delay compared to hardware. In comparison, the results for the hardware-in-the-loop cluster test show much less variability among average send/receive communication times, as shown in Figure 9.

Figure 9 shows a direct comparison between the average send/receive communication times for the hardware-in-the-loop and emulation-in-the-loop clusters.

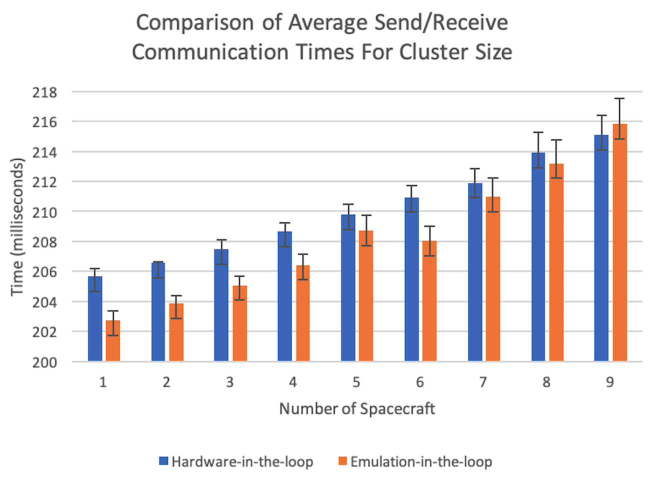


Figure 9. Side-by-side comparison of hardware-in-the-loop and emulation-in-the-loop cluster performance

Additionally, average send/receive communication time for the hardware-in-the-loop cluster also increases linearly with size of the cluster, it increases at a rate of approximately one millisecond per additional spacecraft, whereas the emulation-in-the-loop time increases by a rate of approximately two milliseconds per additional spacecraft. Performing linear regression analysis on these datasets reveals that the hardware-in-the-loop cluster scales with an R^2 value of 0.991 whereas the emulation-in-the-loop cluster scales with an R^2 value of 0.961.

The lower overhead per new spacecraft is expected because of a more distributed computational and communication load; whereas the emulation-in-the-loop cluster deals with additional overhead on the host machine for each additional spacecraft, additional spacecraft in the hardware-in-the-loop cluster have their own compute resources which spares the host computer from the computational load of the flight software.

Embedded Image Processing Test

To perform this test, the frequency of image capture and subsequent processing is fixed to every other image for each platform to assess the effects of additional computation on each platform's performance. This means that every other send/receive cycle also features image capture and processing so it's possible to compare the cycle time with and without image processing. Measurements for send/receive cycles with additional image capture and processing overhead were collected and compared to cycles without this overhead. These results are shown in Figures 10 and 11.

Figure 10 agrees with Figure 9, showing similar average send/receive cycle times of approximately 203 milliseconds and 206 milliseconds respectively. More surprising is the large performance difference for the emulation-in-the-loop and hardware-in-the-loop testbeds shown in Figure 11. Whereas the hardware testbed shows an average send/receive cycle time of approximately 850 milliseconds with a few outliers at 650 milliseconds, the emulation testbed shows much greater variation and a higher overall average of approximately 1400 milliseconds, almost twice the average send/receive cycle time for the hardware testbed, for the same image capture and processing procedure.

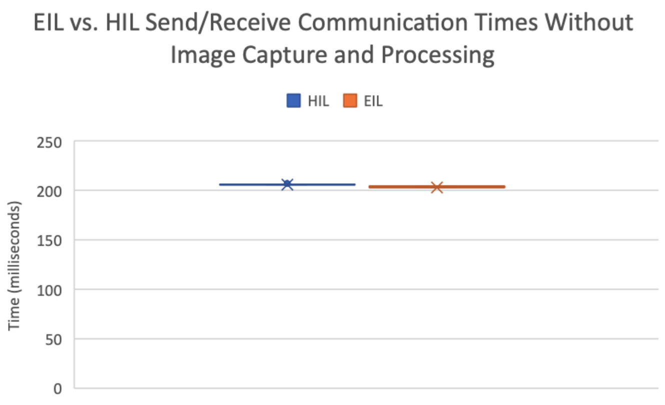


Figure 10. Plot of send/receive communication times in milliseconds for emulation-in-the-loop and hardware-in-the-loop testbeds in the absence of additional computational overhead

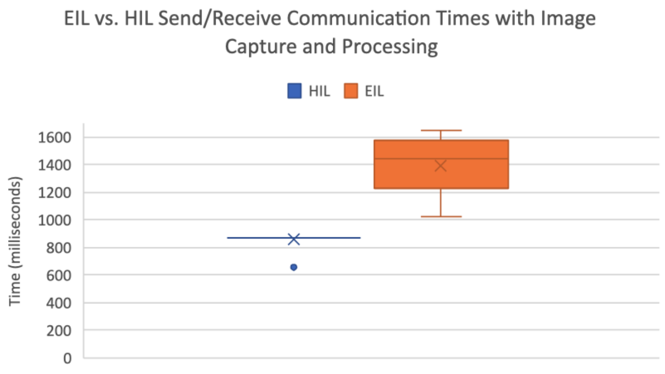


Figure 11. Plot of send/receive communication times in milliseconds for emulation-in-the-loop and hardware-in-the-loop with additional image capture and processing computational overhead

6. Discussion

At a cluster size of 9 spacecraft, the emulation-in-the-loop testbed showed an average send/receive communication time of approximately 215 milliseconds, whereas the emulation-in-the-loop testbed showed a time of approximately 216 milliseconds. This result shows promise for the replacement of hardware with emulation for early-stage small satellite cluster development from the perspective of timing performance. The slowdown in network scalability seen in the cluster scaling test is likely a consequence of the additional QEMU instances slowing down the computer and therefore 42. Thus it is clear that there is great potential for emulation-in-the-loop testing as a replacement or augmentation to hardware-in-the-loop testing for the development of small satellite cluster flight software. The results from the cluster scaling test illustrate that the emulation-in-the-loop testbed offers similar scaling performance to the hardware-in-the-loop testbed. Given the lower cost of the emulation-in-the-loop testbed compared to the hardware-in-the-loop testbed, it's clear that for cluster applications emulation performs well enough to stand in for hardware in terms of timing performance.

The embedded image processing test, on the other hand, shows much improvement to the emulation-in-the-loop testbed is needed to offer similar performance to the hardware-in-the-loop testbed. The large difference in communication times shown in Figure 11 could be explained by several factors. As this test included both a larger socket communication (of an image consisting of approximately 1.92 megabytes) and increased computational complexity as described earlier, this increase could be caused by either factor. It seems that the relatively large socket communication caused these delays, as it is clear from Figure 9 that socket communication overhead and its variation grows faster for the emulation-in-the-loop testbed than for the hardware-in-the-loop testbed. In comparison to the sensor and actuator messages shown in Figures 2 and 3, which are only about 1.94 kilobytes and 0.68 kilobytes respectively, the 1.92-megabyte images are quite significant, requiring multiple send/receive socket calls to send each image.

Overall, there are clear advantages and disadvantages of each testbed. The emulation-in-the-loop testbed is built around QEMU which is a free, open-source tool that offers significantly easier access to testing than acquiring hardware. This stands to reduce mission risk and development time overhead, offering flight software developers early access to the mission hardware environment. On the other hand, this work shows that more investigation is needed to make the emulation-in-the-loop test suitable for more compute- and communication-intensive tasks compared to the hardware-in-the-loop testbed.

7. Conclusions

Hardware remains a critical component for developing and testing flight software for small satellites—it can be expensive and difficult to source. Often, flight software developers do not have early access to hardware on which to test their software due to the high cost of hardware, the fragility of such hardware, and the difficulty of replicating hardware environments at scale. With the growing popularity of hardware emulation and specifically QEMU, emulation stands to augment flight software testing capabilities in the face of the difficulties of working with hardware. This work describes the development of two in-the-loop testbeds—one using emulation, the other using hardware—and how they fare on two different experiments exercising their scalability and computing capabilities. Our results show that the emulation-in-the-loop testbed performs well in terms of scalability, showing similar performance scaling to the hardware-in-the-loop testbed for up to 9 spacecraft. However, the results from the compute-intensive work shows that there is still room for improvement on the emulation-in-the-loop testbed. These results can help inform future small satellite flight software development by demonstrating an emulation-in-the-loop platform and when emulation might be a suitable stand-in for hardware.

8. Future Work

Extensions of this work could include the investigation of different tests, such as the image processing suites presented in [6] and [2], against the two testbeds to evaluate their performance more generally. A formal effort could also be made to evaluate other performance metrics such as memory usage or to compare the capabilities of each testbed for a specific architecture with specific peripherals, as is often necessary for small satellites.

Additionally, potential work exists in improving the performance of the emulation-in-the-loop on the embedded image processing tests, perhaps through the use of alternative methods of communication between the 42 simulator and the QEMU instances (rather than using sockets). For instance, perhaps images could be pre-loaded to eliminate the delay of sending the image, thus estimating just the true computational overhead of performing image processing.

Acknowledgments

This research was supported by the NSF Center for Space, High-performance, and Resilient Computing (SHREC) industry and agency members and by the IUCRC Program of the National Science Foundation under Grant no. CNS-1738783.

References

[1] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41.

[2] Michael J. Cannizzaro, Evan W. Gretok, and Alan D. George. "RISC-V Benchmarking for Onboard Sensor Processing". In: *2021 IEEE Space Computing Conference (SCC)*. 2021, pp. 46–59. DOI: 10.1109/SCC49971.2021.00013.

[3] Katherine Conway et al. *CSP: High Performance Reliable Computing for SmallSats*. Space Micro Inc. Apr. 26, 2017.

[4] Daniel Dvorak. "NASA Study on Flight Software Complexity". In: *AIAA Infotech at Aerospace Conference and Exhibit and AIAA Unmanned...Unlimited Conference* (Apr. 2009). DOI: 10.2514/6.2009-1882.

[5] Antony Gillette, Alan George, and J. Patrick Castle. "Design and Validation of an Autonomous Mission Manager towards Coordinated Multi-Spacecraft Missions". In: (2021).

[6] Evan W. Gretok and Alan D. George. "Onboard Multi-Scale Tile Classification for Satellites and Other Spacecraft". In: *2021 IEEE Space Computing Conference (SCC)*. 2021, pp. 110–121. DOI: 10.1109/SCC49971.2021.00019.

[7] Matthew D. Grubb. "Increasing the Reliability of Software Systems on Small Satellites Using Software-Based Simulation of the Embedded System". In: (2021).

[8] David McComas, Jonathan Wilmot, and Alan Cudmore. "The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft". In: 2016.

[9] Justin Morris et al. "Simulation-to-Flight 1 (STF-1): A Mission to Enable CubeSat Software-based Verification and Validation". In: *54th AIAA Aerospace Sciences Meeting*. DOI: 10.2514/6.2016-1464.

[10] Paulo Renato Oliveira et al. "Emulation-in-the-loop for simulation and testing of real-time critical CPS". In: *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*. 2018, pp. 258–263. DOI: 10.1109/ICPHYS.2018.8387669.

[11] Gyokan O Osman. "Emulating the Internet of Things with QEMU". In: (2020).

[12] *Q7 Processor*. Xiphos Technologies, 2021.

[13] Dylan Rudolph et al. "CSP: A multifaceted hybrid architecture for space computing". In: (2014).

[14] Eric Stoneking. "An Introduction to Simulation Using 42". In: (2017).

Biography

Rachel Misbin *received her B.S. degree in computer engineering from the University of Pittsburgh. She is pursuing her M.S. degree in electrical and computer engineering from the University of Pittsburgh with the NSF SHREC Center. She has previously interned at Blue Origin and The Kathrine Johnson Independent Verification and Validation (IV&V) Facility at NASA working on projects related to flight software development and testing.*

Alan George *is Department Chair and the R&H Mickle Endowed Chair in Electrical and Computer Engineering in the Swanson School of Engineering at the University of Pittsburgh. He founded and directs the NSF Center for Space, High-performance, and Resilient Computing (SHREC), which replaced the NSF Center for High-performance Reconfigurable Computing (CHREC) in late 2017. Dr. George's research interests are in advanced architectures, apps, networks, services, systems, and missions for reconfigurable, parallel, distributed, and dependable computing. He is a Fellow of the IEEE.*